Self-testable components: from pragmatic tests to a design-for-testability methodology

Daniel Deveaux [1], Jean-Marc Jézéquel [2] and Yves Le Traon [2]

[1] VALORIA, Université de Bretagne, Campus de Tohannic 56000 Vannes Cedex, France

Daniel.Deveaux @univ-ubs.fr

[2] IRISA-INRIA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France

{yletraon, Jean-Marc.Jezequel}@irisa.fr

Abstract: Testing is a key aspect of software development, because of its cost and impact on final product reliability. Classical views on testing and their associated testing models, based on the waterfall model, are not well-suited to an OO development process. The standardization of semi-formal modeling methods, such as UML, reveals this trend: testing can no longer be separated from specification/design/code stages. A test approach integrated with the OO process must be defined with an associated testing philosophy.

The approach presented in this paper aims at providing a consistent framework for building trust into components. By measuring the quality of test cases, we seek to build trust in a component passing those test cases. We present a pragmatic approach for linking design and test of classes, seen as basic unit test components. Components are self-testable by enhancing them with embedded test sequences and test oracles. Self-testable components serve as building blocks for performing systematic integration and non-regression testing. The main contribution presented in this paper consists of using component self-tests to systematically exercise main system structural dependencies.

This approach has been implemented in the Eiffel, Java, Perl and C++ languages. Since it is simpler, due to the direct support for Design-by-Contract™ in the language, the Eiffel implementation is detailed here.

I. Introduction

Using "off-the-shelf" software components in mission-critical applications is not yet as commonplace as in desktop applications. Opposite to electronic components, there is the lack of method and quality estimates in the software domain which provide reasons to trust software components. While electronic devices have set of measures characterizing their quality (reliability, performances, use-domain, speed scale), no real consensus exists to measure such quality characteristics for software components. The second problem is the software component expected capability to evolve (addition of new functionality, implementation change) and to be adapted to various environments. As for hardware systems, we propose to build a trust on components through testing. In the case of OO development, classical views on testing and their associated testing models are not well-suited to the significant changes this software paradigm has induced on the development process [Binder94]. The standardization of semi-formal modeling methods, such as UML, reveals this trend: testing can no longer be separated from specification/design/code stages. A test approach integrated with the OO process must be defined with an associated testing philosophy.

In this paper, we present a pragmatic approach for linking design and test of classes, seen as basic unit test components. Noting that the feasibility of class testing is similar to classical procedural testing, each component is enhanced by the ability to invoke the unit test by itself: components are made *self-testable*.

We consider a self-testable component as a set containing a specification, a test suite and a given implementation. To build trust on a component, we propose to estimate the quality of its test sequence. So, the self-testable component carries an associated value, a quality metric, which quantifies the quality of the unit test sequence for testing the given implementation with respect to the reference specification. This quality metric is provided by selective mutation analysis, which has been adapted to OO languages.

Once a component has reached a given level of trustability, self-test still serves as a basis for performing systematic integration and non-regression testing.

The rest of the paper follows with a discussion on the testing process and the software attributes which have an impact on OO testing (Section 2). Section 3 is devoted to the definition of self-testable components. This is illustrated with an implementation with Eiffel. Section 3 defines the notion of test quality and shows how the programmer can improve it. Section 5 concentrates on the definition of structural test dependencies which serve

as a basis for defining system test strategies guided by system coverage. Related works are presented in Section 6.

II. The Process of Software Testing

The purpose of testing is to find out whether there are faults in a software system (in the IEEE standard terminology, *errors* represent human mistakes that can result in *faults* or *bugs* in a system; the execution of a faulty system produces *failures*). The process of testing is to find the minimum number of test cases that can produce the maximum number of failures in the concerned system to achieve a desired level of confidence.

Exhaustive testing is generally not possible, the only result that can be expected is a high level of confidence that the unit being tested operates correctly. A strategy is thus needed to efficiently drive the testing process. The testing activity can roughly be divided into five stages that are still applicable in an object-oriented context:

1. Identification of the *testing criteria*. Testing criteria define the goals for comparing a system to a specification. In classical software testing, a testing criteria is basically defined as the coverage of an abstract view of the software (for white-box testing, an all nodes coverage in a control-flow graph or an all definition-use paths coverage in data-flow graph [Rapps85]). A testing criterion serves as a reference for deciding if actual tests achieve testing requirements. To define a testing criterion adapted to OO paradigm, such an abstract view of an OO system has to be proposed as a good modeling of most significant testing attributes. One of the objectives of this paper is to propose a model of structural test dependencies that catches all main test significant information. Test strategies are specified relatively to coverage criteria of the model.
2. Identification of the *target components* for testing. In procedural systems, the components that were subject to testing were procedures and functions, clusters of procedures and functions, execution paths, and the complete application as a whole. This corresponds to a hierarchy of testing into unit, integration and system test stages. In this paper, we claim that making distinctions between integration and systems testing is artificial, due to the "organic growth" of OO systems. Indeed, any class and its methods may be considered as defining some kind of "unit component", assuming that its inherited classes are tested separately. However, the OO process leads to a progressive and recursive life cycle where integration and system tests are indistinctly performed. For building a functionally coherent system, one needs various steps such as putting components into packages, using/reusing and assembling them together into an inheritance or client hierarchy. Moreover, such an OO system has generally no simple well demarcated frontier and some of its parts may always evolve or be reused separately. There are thus many reasons for improving the way of trusting components and packages.
3. *Generation of test case and oracles*, where each test case consists of test input (test case or sequence of test cases) and expected output (an "oracle" function). Test case generation is often driven by the testing criteria presented above. Of all the stages of testing, test case generation is the most time consuming because it is not so prone to automation (non-automated test generation is said to be "deterministic"). The primary goal of test generation is to produce test cases that can identify faults in an implementation. The number of admissible test cases for a given design of a program is huge, so the actual goal of test case generation is to generate the least number of test cases that are effective in identifying most of the faults in a program. In the case of OO software, each class is considered as a well functionally defined basic test unit. In this well defined testing domain, classical procedural selection criteria always work. In this paper we suggest to use mutation testing as a guidance both for unit testing criteria and testing quality assurance.
4. *Execution of test cases* against the target components. Once the tests have been defined, it may be necessary to develop specific programs or environments (stubs and drivers) to run the tests. Being given testable classes, the question is thus to be able to execute automatically unit class testing to perform unit test and, if possible, some kind of non-regression and integration testing.
5. Evaluation to determine whether the tests have produced the expected results. If not, a bug report has to be issued. This paper borrows from [Jézéquel96] the idea of using assertions as test oracles. Assertions are executable elements of formal specifications, as available in e.g.; Eiffel. Most interesting here are post-conditions, class invariants, loop variants and invariants.

The quality attribute of the software which is related to this process of testing is called testability. Testability is a property of both the software and the process and refers to the "easiness" for applying all the previous steps and on the inherent of the software to reveal faults during testing ([Voas92, LeTraon97]). Two way of appraising and improving testability may be envisaged depending if we focus on the inherent software testability or on the process of testing. In this paper, the process of software testing is organized from unit testing to some structural system test into a global and structured testing methodology: a *design for testability* approach is thus proposed ([Binder94]).

III. Self-testable OO components

In this section, we define self-testable component before detailing test cases generation and the way of evaluating component test quality.

III.1. Aims and definition

We propose the triangle view for designing, as illustrated in Figure 1. Indeed, due to the life and possible evolution of a software component, a sort of organic link must be maintained between the specification, the test set and the chosen implementation. The methodology is based on a integrated design and test approach for OO software components. Classes are considered as basic unit components. Test suites are defined as being an "organic" part of software OO component. Indeed, a component is composed from its specification (documentation, methods signature and invariant properties), one implementation and the test cases needed for testing it. This view of an OO component is illustrated under the triangle representation (cf. Figure 1). To a component specified functionality is added a new feature which enables it to test itself: the component is made *self-testable*.

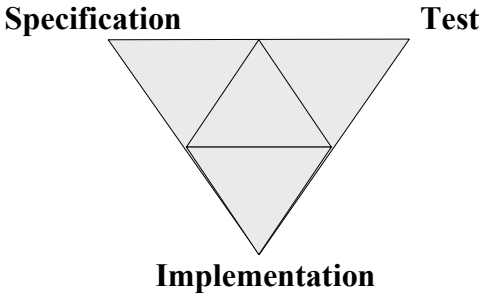

Fig. 1. The triangle view of a component

Based on this view, it is then the class implementor's responsability to ensure that all the embedded tests are satisfied. So, one can estimate the *test quality* relatively to the specification, a test sequence and a given implementation. As soon as the quality level is not reached, the test sequence must be enhanced. So when used, a self-testable component may test itself with a guaranteed level of quality. This quality level could be defined under several ways (such as classical definition-use coverage): in this paper mutation analysis [Offutt92] is proposed as a relevant way for analyzing the quality of a the test sequence. Quality measurement is thus defined based on the fault revealing power of the test sequence when systematic fault injection is performed. Once such a test quality estimate is associated to a set of functionally-equivalent components, the designer can choose the component with the best self-test ability.

The execution of the component test suite might trigger the creation of many other objects (from which the component is a client) end/or require the execution of inherited features. This shows that a class under test is in test relationship with its clients and parents. Such relationships correspond to structural test dependencies. As these direct dependencies are only partially tested by the execution of class-under-test self-tests, one may use class self-test-ability to systematically test parts or of whole the system structural dependencies. For example, in the case presented in Figure 2, if the component C1 is modified (change in the implementation), at least all its direct contractual dependencies with the components C3 and C4 it uses have to be tested. Also, because C2 is a client of C1, C2 needs to execute its own self-test for non-regression. Such dependencies do not depend on the components themselves but on the system architecture. The next section details such an use at system level of local components self-tests. Guided structural system test strategies and adequacy criteria are presented to retest the system and assert non-regression when implementation or specification changes are performed.

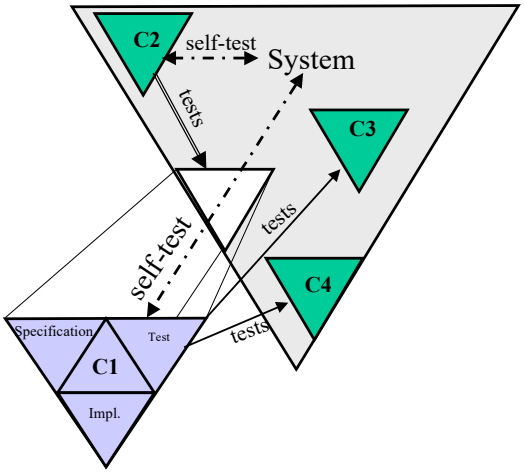

Fig. 2 Self-tests for system structural tests

III.2. Example implementation with Eiffel

This approach has been implemented in the Eiffel, Java, Perl and C++ languages. Since it is simpler, due to the direct support for Design-by-Contract™ in the language, the Eiffel implementation is detailed here. All the details of these implementations are available from http:/www.irisa.fr/testobjets/self-test.

How to make a self-testable class

The aim of this Section is to show how to evolve an Eiffel class into an self-testable one. For the sake of simplicity, we take the example of a trivial implementation of a bounded set of integers whose interface is presented in Figure 3.

```
Interface of class SET_OF_INTEGERS
 --  implements a simplified set of integers.
 --  This class is developed for
 --  demonstration purpose only

creation     make

feature -- creation

    make
      -- allocate a set of integers
            ensure          empty: empty;
feature {ANY} -- status report

    empty: BOOLEAN
      -- is the set empty?

    full: BOOLEAN
      -- is the set full ?

feature {ANY} -- access

    has(x: INTEGER): BOOLEAN
      -- is x in the set ?

feature {ANY} -- Element changes

    put(x: INTEGER)
      -- put x into the set
            require         not_full: not full;
            ensure          has: has(x);
                            not_empty: not
empty;

    prune(x: INTEGER)
      -- remove x from the set.
            require         has: has(x);
                            not_empty: not
empty;
            ensure          not_has: not
has(x);
                            not_full: not full;

 end --  class SET_OF_INTEGERS
```

Fig. 3. Set_of_integer: an Eiffel interface

The purpose of the self-test is to call (even indirectly) all the testing routines of its class. To make this class self-testable we add a inherit SELF_TEST clause and we declare the inherited method test as creation feature (constructor).

```
inherit
  SELF_TESTABLE;

creation {ANY}
  make, self-test
```

The class SET_OF_INTEGER has any methods that should be all called and verified. Each testing routine has a goal (explain in his comment) which is to test that the implementation of a set of methods corresponds to

their specification. By convention, the testing method name begin with tst_. SET_OF_INTEGER is a very simple example, only two testing methods are defined (tst_add and tst_prune, the later being presented in Figure 4). In a real class, more methods should probably be needed to test all the class features. The parent class SELF_TEST define a set of useful functions (test_title, test_msg, ...) that help the messages writing in testing methods.

```
tst_prune is
      -- to test the relialabity of `prune' and `has'
      do
        test_title("add and remove","put - has - prune");
        make;
          -- known state of the set
        if not full then          put(1);       end;
        if not full then          put(2);       end;
        if not full then          put(3);       end;
        test_msg("add 3 elements ... Ok");
          -- to remove all the elements independantly of the order
          -- in which they were put.
        check          has_3: has(3);       end;
        prune(3);
        check
          not_full: not full;
          not_empty: not empty;
        end;
        test_msg("remove '3' ... Ok");
        check          has_1: has(1);       end;
        prune(1);
        test_msg("remove '1' ... Ok");
        check          has_2: has(2);       end;
        prune(2);
        check          empty: empty;        end;
        test_msg("remove '2', empty set ... Ok");
      end -- tst_prune
```

Fig. 4. Test of the prune method

```
feature {SET_OF_INTEGERS}   -- self-test sequence

   test_seq is
     -- call test routines in sequence
     do
       if test_make(1,"add")          then tst_add          end
       if test_make(2,"prune")        then tst_prune        end
     rescue
       test_error;
       retry;
     end -- test_seq
```

Fig. 5.Global self-test sequence for testing add and prune

The method test_seq, presented Figure 5, arranges the test sequence; it is invoked by the inherited test launcher self-test. The method test_make controls the testing method execution, depending on runtime parameters.

The rescue clause allows the self-test to continue on the next test after an error is detected and a precise message is provided (this is the role of the test_error method). This class can be compiled as a stand alone program, using the method selftest as its entry point. The output of this program is presented in Figure 6.

```
Test of class SET_OF_INTEGERS

------------------------------------------------
  Test sequence nb. 1 is add usable?

        put - empty - full
        ------------------
        - empty at create ... Ok
     - not empty after put ... Ok
     - add 10 elements ... Ok

------------------------------------------------
   Test sequence nb. 2 add and remove

      put - has - prune
      -----------------
      - add 3 elements ... Ok
      - remove '3' ... Ok
      - remove '1' ... Ok
      - remove '2', empty set ... Ok

------------------------------------------------

>>>> DIAGNOSIS on class SET_OF_INTEGERS <<<<
 test(s) are OK.

------------------------------------------------

  Method call statistics
      has              : 37
      full             : 33
      prune            : 3
      empty            : 47
      index_of         : 40
      put              : 14
```

Fig.6. A self-test synthetic execution log

By default, the validation test is run using options in the command line, it is possible to control debugging functions (level of profiling) and to select which testing methods one wants to run.

The execution logs can be redirected to files. This instrumentation is useful not only for the validation phase, but also for verification and debugging.

IV. Test quality

A test quality estimate can be associated to each self-test to provide a guide in a choice of a component, and to help reaching the test adequacy criteria. This quality estimate quantifies the trust one can have in a tested component. The chosen quality criteria proposed here is the proportion of injected faults the self-test detects when faults are systematically injected into the component implementation. This estimate is, in fact, derived from the mutation testing technique, which is adapted for OO classes.

IV.1. Mutation testing technique for OO domain

Mutation testing is a testing technique which is first designed to create effective test data, with an important fault revealing power [Offutt96, Voas92]. It has been originally proposed in 1978 [DeMillo78] , and consists in creating a set of faulty versions or *mutants* of a program with the ultimate goal of designing a test set that distinguishes the program from all its mutants. In practice, faults are modeled by a set of *mutation operators* where each operator represents a class of software faults. To create a mutant, it is sufficient to apply its associated operator to the original program.

A test set is relatively adequate if it distinguishes the original program from all its non-equivalent mutants. Otherwise, a *mutation score(MS)* is associated to the test set to measure its effectiveness in terms of percentage of the revealed non-equivalent mutants. It is to be noted that a mutant is considered equivalent to the original program if there is no input data on which the mutant and the original program produce a different output. A benefit of the mutation score is that even if no error is found, it still measures how well the software has been tested giving the user information about the program in the absence of errors. It can be viewed in a way as a reliability assessment for the tested software.

In this paper, we are looking for a subset of mutation operators :

- general enough to be applied to various OO languages (Java, C++, Eiffel etc)

- implying a limited computational expense,
- ensuring at least control-flow coverage of methods.

The choice of mutation operators is given in the appendix . Our current choice of mutation operator include selective relational and arithmetic operator replacement, variable perturbation but also referencing faults for declared objects. During the test selection process, a mutant program is said to be *dead* if at least one test case detects the fault injected into the mutant. Conversely, a mutant is said to be *alive* if no test cases detects the injected fault.

IV.2. Test selection process

The whole process for generating unit test cases with fault injection is the following:

1) Initialization

a) generate a first set TC of nTC test cases,
b) generate the set of mutants M,
c) choose the wanted quality level of the test suite expressed as an expected mutation score EMS,
d) choose a maximum number of test cases MaxTC (needed for progressive equivalent mutant detection)

2) Mutation analysis and test cases enhancement process

While MS < EMS and nTC< MaxTC do

a) enhance the test cases set, update nTC,
b) apply each new test case on each alive mutant,
c) update the sets of alive and dead mutants,
d) compute the new mutation score MS.

3) Determination of equivalent mutants

If nTC = MaxTC then

a) human analysis is needed for determining if equivalent mutants exists,
b) suppress equivalent mutants from the set of alive mutants,
c) choose a new MaxTC value
d) re-apply on point 2.

4) Test cases selection process:

Choose a subset of T which preserve the mutation score: suppression of redundant test cases.

It has to be noted that when the set of test cases is selected, the mutation score is fixed as well as the test quality of the component. Moreover, except for step 2a and 3a, the process can be completely automated.

IV.3. Test cases generation and oracle determination

Deterministic test data generation is used since each class is made of a a set of functionally coherent methods. Basic efficient data are easy to generate for designer and developer. Indeed, experience teaches that deterministic test generation can follow the following rules:

1) Methods which are functionally linked belong to a same family of testing. They have to be tested together (like for a set; *prune* and *put,* since you cannot test *prune* without putting one element in the set).
2) In a family, basic independent sets of methods have to be exercised first. For example, *has* cannot be tested before *put* but also you need *has* to check if *put* is correct. So *has* and *put* must be tested together first, before prune which is less basic and needs both *has* and *put* to be tested.
3) In case of inheritance, redefined methods have to be re-tested with specific tests.

For generating oracles, the most general solution consists of writing explicit test oracles for each test suite. For example, designer knows that a [put(2), prune(2)] test sequence implies that [not has(2)] should return a *true* value. The oracle is thus simply obtained by checking that 2 is not present in the set of integer. To be coherent with the rules expressed upward, the method *has* should have been tested before this test suite is exercised. The second way is adapted to a design-by-contract approach. In this approach, post-conditions, i. e. invariant expressions on the output domain values and relationships with the input ones, are used as partial oracle functions. Most of faults can be detected in a systematic design-by-contract approach without writing explicit oracle functions. Post-conditions are thus covering a larger space of test data but are generally not sufficient for detecting particular semantically rich test results. In some cases, the post-condition is sufficiently precise to replace deterministic oracles: for a sort method, testing if the result is effectively sorted is a complete and simple-to-express oracle function. However, in most cases, functional dependencies between methods are difficult to express through general invariants.

IV.4. Component and system test quality

The test quality of a component is simply obtained by computing the mutation score for the unit testing test suite executed with the self-test method.

The system test quality is defined as follows:

Let S be a system composed of n components denoted C_i , $i \in [1..n]$,

let d_i be the number of dead mutants after applying the unit test sequence to C_i, and m_i the total number of mutants.

The test quality,i. e. the mutation score MS, for C_i being given a unit test sequence T_i is defined as follows :

$$TQ(C_i, T_i) = \frac{d_i}{m_i}$$

And the System Test Quality (STQ) is defined relatively to the d_i and m_i as follows :

$$STQ(S) = \frac{\sum_{i=1}^{n} d_i}{\sum_{i=1}^{n} m_i}$$

These quality parameters are associated to each component and the global system test quality is computed and updated depending on the number of components actually integrated to the system.

V. System self-test strategy

In this section, each component self-test serves as a basis for performing systematic integration and non-regression at system level. The main idea presented here consists in using components self tests to systematically exercise the system main test structural dependencies.

V.1. Modeling system structural test dependencies

At system level, two categories of test sets may be foreseen with respect to a component: *implementation-dependent test* sets and *implementation independent* ones. Implementation-dependent test (IDT) set is a set of test data and oracles which uses information specific to the implementation (access to internal or private variables and methods) while implementation independent test (IIT) sets may be executed on any implementation target. Implementation independent test cases correspond to black-box testing. For a component C into a system S, three main possible cases may be encountered:

- *modification of C implementation*: the IIT self-test method allows to verify the new implementation correctness, specification does not change, IDT has to be re-written,
- *addition of new functionality to C* : IIT and IDT self-tests must be executed to assert non-regression testing and then completed to test new functionality. Specification must be updated.
- *reusing/integration of C into S:* for integrating a component or re-integrating it after modification/evolution, a global system strategy must be used with respect to a chosen adequacy criterion. The global strategy will drive the execution of the self-tests for each component dependent from C. Next section discusses both non-regression and integration with respect to self-testability.

V.2. Test dependencies

Let Ci and Cj be two components of a system S, the concept of system test dependency (an intuitively natural concept) is defined as follows:

Test dependency : A component class C_i is *test-dependent* from Cj if it uses some objects from C_j or inherits from C_j. This dependency is noted:

$$C_i \; R_{TD} \; C_j$$

Contractual Dependency (CD): A component C_i is *contractual-dependent* from C_j if it uses C_j or inherits from C_j, whatever the implementation choices are. We call a *Inheritance Contractual Dependency (ICD)*, a for contractual dependency where C_i inherits from Cj and *a Client Contractual Dependency (CCD)* a contractual dependency such as C_i declares at least one object from Cj. Contractual dependency is noted $C_i \; R_{CD} \; C_j$ and is thus decomposed into inheritance dependency (noted $C_i \; R_{ICD} \; C_j$) or client one ($C_i \; R_{CCD} \; C_j$).

Implementation Dependency (ID): A component C_i is implementation-dependent from Cj if: C_i R_{TD} C_j and not C_i R_{CD} C_j. It is noted C_i R_{ID} C_j

Test Dependency Graph : It is a directed graph whose nodes represent components and whose transitions represent dependencies. Three types of transitions are taken into account corresponding to the three types of previously defined test dependencies (ICD, CCD and ID). If information are known about the methods of a class which are involved in the dependencies, the transitions may be labeled with the names of the methods. In such a test dependency graph, loops may occur because components may be directly or indirectly test dependent from each other.

Non-regression test adequacy criteria: For testing a component C into a system, two basic criteria may be used. The weakest consists of executing self-tests of components which are directly dependent from C (direct-dependencies coverage). The strongest consists of executing self-tests for each component which is included into a path containing C (all-dependencies coverage). The notion of direct and all dependency adequacy criteria may be applied to each type of dependency (contractual inheritance, contractual client and implementation dependencies). A simple partial order relationship exists between adequacy criteria as presented for contractual dependencies in Figure 7 (the stronger criterion is on the top of the figure).

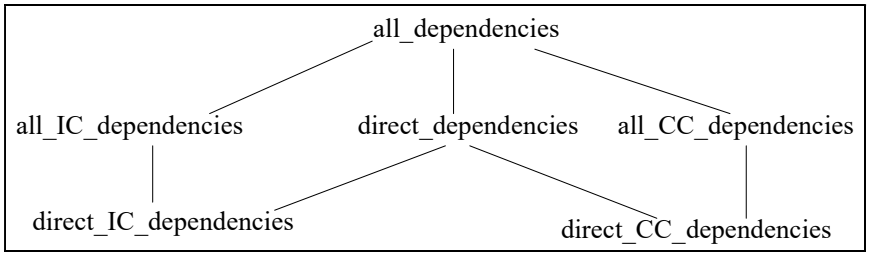

Fig. 7. Partial order between test adequacy criteria

The way of building a test dependency graph is illustrated in Figure 8. In this example, the model is simply deduced from a UML static class diagram. Aggregation, composition navigability and dependency relations, and other basic operations on class diagrams create client contractual dependencies (CCD) into the corresponding test dependency graph.

Integration testing : The integration strategy is based on the decomposition of the test dependency graph. As a result, the components are ordered from the easier to test to the most difficult one, with respect to the number of *stubs* to be created. The algorithm proceeds by decomposing the graph into its strongly connected component (existing loops are broken) and organizing the test by minimizing the stubs to be written. Indeed, the *test driver* is not a problem since the self-test for each component has just to be called.

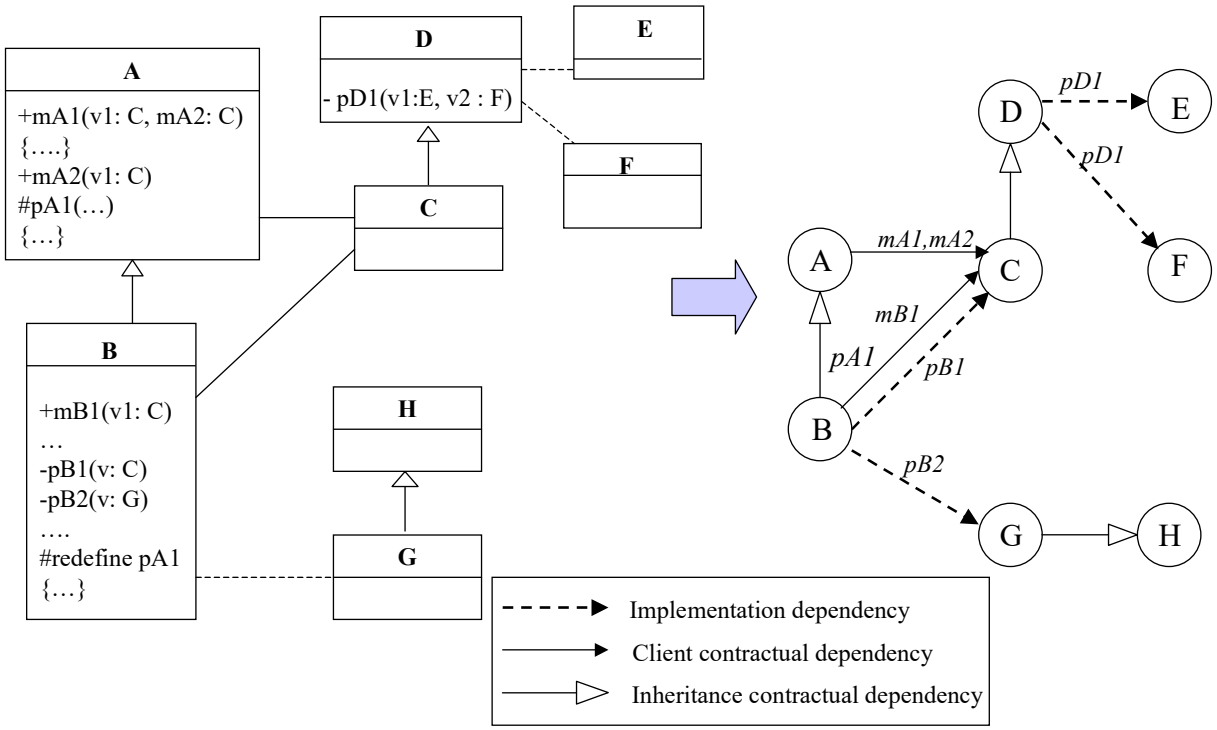

Fig. 8. A UML description and its associated test dependency graph

V.3. Designing Self-testable Hierarchies

The test of large classes library should be automated through a **test driver**. We have implemented a generic test driver written in Perl to activate self-testable classes in various languages (actually Perl, Eiffel and Java). This driver perform the following functions:

- identify the self-testable classes in the directory hierarchy,
- compile classes in test mode,
- run each class test committing the detailed testing report,
- compose a short synthetic testing report

This driver uses a simple configuration language defining the test strategies. The practical application of the modeling techniques presented above allows the automatic production of the driver script.

VI. Related works

Very few of the numerous first-generation books on analysis, design, and implementation of object-oriented software explicitly address V&V issues. Despite this initial lack of interest, testing of object-oriented systems is receiving much more attention (see [Binder96] for a detailed state of the art).

Concerning OO testing techniques, most of the works focus on the dynamic aspects of OO systems: a system is viewed as a set of cooperating agents, modeling objects, and modeled with FSM, or equivalent object-state modeling [Binder94, Jorgensen94, Kung96]. Such works have to deal with limitations concerning computational expense of mapping objects behaviors into the underlying model. One solution consists in decomposing the program into hierarchical and functionally coherent parts. In such approaches, this decomposition provides a framework for unit, integration and system test definition. In [McGregor94], the waterfall model is overtaken and an integrated test and development approach is proposed. These state-based models constrain the design methodology to divide the system into small parts with respect to behavioral complexity. Binder details the existing analogy between hardware and OO software testing and suggests an OO testing approach close to the "built-in-test" and "design-for-testability" hardware notions. In this paper, we go even further than Binder suggests, and detail how to create self-testable OO components, with an explicit analogy with the "built-in-self-test" hardware terminology. Concerning test strategies, our work is very much along the lines of [Harrold92, Kung96] approaches. In particular, Kung proposes a method for identifying affected classes during maintenance and giving a desirable order to test these classes. The used object relation graph model serves for ordering

classes to be tested for non-regression, integration and maintenance purposes. The main difference is that our approach, being more modular, is more suited to self-testable components.

Besides, the test problem may be seen from a pragmatic point of view, and some simple-to-apply methodology can be found in the literature, which are based on an explicit test philosophy [Beck98]. In this paper, the proposed methodology is based on pragmatic unit test generation and aims at bridging the existing gap between unit and system dynamic tests. The notion of structural test dependencies has been developed for modeling the systematic use of self-testable components for structural system test. Moreover, an original measure of the quality of components has been defined based on the quality of their associated tests (itself based on fault injection). For measuring test quality, the presented approach differs from classical mutation analysis [Offutt96, DeMillo91] as follows:

- a reduced set of mutation operators is needed,
- oracles functions are integrated to the component, while classical mutation analysis uses differences between original program and mutant behaviors to craft a pseudo-oracle function.

VII. Conclusion

The approach presented in this paper aims at providing a consistent framework for building trust into components. By measuring the quality of test cases (the revealing power of the test cases [Voa92]) we seek to build trust in a component passing those test cases. The production of self-testable components has been detailed as well as a methodology for ensuring test quality and systematic non-regression and integration testing. The approach is based on the following considerations:

- writing tests is easy at a unit class level [Beck98],
- verifying the test quality is feasible through fault injection. Mutation may also be used to select a reduced set of test cases. The process of estimating test quality can be automated,
- since test driver consists in a set of self-test method calls, structural system tests are easy to launch.

Based on these considerations, a model of structural system test dependencies has been defined. It serves as a basis for guiding system self-test. Depending on the validation context (non-regression or integration) test adequacy criteria have been proposed. Further work will detail experimental studies for validating the relevance of mutation operators to an OO context and integration strategies based on the underlying test dependency model.

References

[Beck98] K. Beck and E. Gamma, "Test-Infected: Programmers Love Writing Tests,"Java Report, July1998, 37-50.

[Binder94] Robert V. Binder, "Design for Testability with Object-Oriented Systems," Communications of the ACM, v 37, n 9, September 1994, 87-101.

[Binder96] Robert V. Binder. Testing object-oriented software : A survey. Journal of Sofware Testing, Verification and Reliability, 6(125-252), 1996.

[DeMillo78] R. DeMillo, R. Lipton, and F. Sayward, “Hints on Test Data Selection : Help For The Practicing Programmer,” IEEE Computer, vol. 11, pp. 34-41, 1978.

[DeMillo91] R. DeMillo et A. Offutt, “Constraint-Based Automatic Test Data Generation”, IEEE Transactions On Computers, vol. 17, pp. 900-910, 1991.

[Harrold92] Mary Jean Harrold, John D. McGregor, and Kevin J. Fitzpatrick, "Incremental Testing of Object-oriented Class Structures," Proceedings, 14th International Conference on Software Engineering, May 1992. IEEE Computer Society Press, Los Alamitos, Calif. 68-80.

[Jézéquel96] Jean-Marc Jézéquel. Object Oriented Software Engineering with Eiffel. Addison-Wesley, mar 1996. ISBN 1-201-63381-7.

[Jorgensen94] Paul C. Jorgensen and Carl Erickson, "Object-Oriented Integration Testing," Communications of the ACM, v 37, n 9, September 1994, 30-38.

[Kung96] David C. Kung, Gao, Jerry, Chen, Cris. "On Regression Testing of Object-Oriented Programs," The Journal of Systems and Software. Jan 1996 v 32 n 1.

[LeTraon95] Y. Le Traon et C. Robach, “From Hardware to Software Testability”, presented at IEEE International Test Conference (ITC’95), Washington D.C, 1995.

[LeTraon97] Y. Le Traon and C. Robach, “Testability Measurements for Dataflow Designs,” Proc. of the 4th International Software Metrics Symposium (Metrics 97), Albuquerque (New Mexico), 1997, pp. 91-98

[McGregor94] John D. McGregor and Tim Korson, "Integrating Object-Oriented Testing and Development Processes," Communications of the ACM, v 37, n 9, September 1994, 59-77.

[Meyer92] B. Meyer. Applying ``design by contract". IEEE Computer, pages 40--51, oct 1992.

[Offutt92] A. J. Offutt, “Investigation of the software testing coupling effect”, ACM Transaction on Software Engineering Methodology, vol. 1, pp. 3-18, 1992.

[Offutt96] J. Offutt, J. Pan, K. Tewary and T. Zhang "An experimental evaluation of data flow and mutation testing," Software Practice and Experience, v 26, n 2, February 1996.

[Rapps85] S. Rapps et E. J. Weyuker, “Selecting Software Test Data Using Data Flow Information”, IEEE Transactions on Software Engineering, vol. 11, pp. 367-375, 1985.

[Voas92] J. Voas et K. Miller, “The Revealing Power of a Test Case”, Software Testing, Verification and Reliability, vol. 2, pp. 25-42, 1992.

Appendix : Mutation operators

Type	Description
EHF	Exception Handling Fault
AOR	Arithmetic Operator Replacement
LOR	Logical Operator Replacement
ROR	Relational Operator Replacement
NOR	No Operation Replacement
VCP	Variable and Constant Perturbation
RFI	Referencing Fault Insertion

Table 1: Mutation operators set for OO programs

Description of the functionality of each of the mutation operators:

EHF: Causes an exception when executed. This semantically large mutation operator allows to force code coverage.

AOR: Replaces occurrences of "+" by "-" and vice-versa.

arithmetic operator	replaced by
+	-, *
-	+, / (or div)
*	/ (or div), +
/	*, -
div	-, mod
mod	-, div

LOR: Each occurrence of one of the logical operators (*and*, *or*, *nand*, *nor*, *xor*) is replaced by each of the other operators; in addition, the expression is replaced by TRUE and FALSE.

ROR: Each occurrence of one of the relational operators (<, >, <=, >=, =, /=) is replaced by each one of the other operators. To avoid semantically large mutation the following rules are applied:

relational operator	replaced by
<	<=
>	>=
/=	< and >
<=	<
>=	>
=	>= and <=

NOR: Replaces each statement by the *Null* statement.

VCP: Constant and variables values are slightly modified to emulate domain perturbation testing. Each constant or variable of arithmetic type is both incremented by one and decremented by one. Each *boolean* is replaced by its complement.

RFI: Stuck-at void the reference of an object after its creation. Suppress a clone or copy instruction. Insert a clone instruction for each reference affectation.